# Breaking Elliptic Curve Cryptosystems using Reconfigurable Hardware

Junfeng Fan[†], Daniel V. Bailey[‡*] , Lejla Batina[†◁], Tim Güneysu[‡], Christof Paar[‡] and Ingrid Verbauwhede[†]
[†] ESAT/SCD-COSIC, Katholieke Universiteit Leuven and IBBT
Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium
{*Firstname.Lastname*}*@esat.kuleuven.be*
[‡] Horst Görtz Institute for IT Security, Ruhr University Bochum, Germany
{*gueneysu, cpaar*}*@crypto.rub.de*
[*] RSA, the Security Division of EMC, USA
*dbailey@rsa.com*
[◁] Radboud University Nijmegen, Netherlands

***Abstract*—This paper reports a new speed record for FPGAs in cracking Elliptic Curve Cryptosystems. We conduct a detailed analysis of different $\mathbf{F}_{2^m}$ multiplication approaches in this application. A novel architecture using optimized normal basis multipliers is proposed to solve the Certicom challenge ECC2K-130. We compare the FPGA performance against CPUs, GPUs, and the Sony PlayStation 3. Our implementations show low-cost FPGAs outperform even multicore desktop processors and graphics cards by a factor of 2.**

***Index Terms*—Elliptic Curve Cryptography; Certicom Challenge; FPGA;**

## I. Motivation: Attacking ECC2K-130

Elliptic-Curve Cryptosystems (ECC), independently invented by Miller [18] and Koblitz [15], are now commonplace in both the academic literature and practical deployments. ECC allow shorter key-lengths, ciphertexts, and signatures than other conventional cryptosystems, *e.g.*, RSA, and thus admit valuable optimizations in computing and communication complexity.

The security of ECC relies on the difficulty of Elliptic Curve Discrete Logarithm Problem (ECDLP) [3]. Briefly speaking, ECDLP is to find an integer $n$ for two points $P$ and $Q$ on an elliptic curve $E$ such that $Q \equiv [n]P$. To use ECC in the real world, practitioners need to know: how big should the parameters (or, colloquially, the "key size") be to avoid practical attacks? Choosing parameters too small allows computational attackers to solve the ECDLP instance, while choosing parameters too large wastes time, communication, and storage. To encourage investigation of these issues, researchers at Certicom Corp. published a list of ECDLP challenges in 1997 [7].

Smaller members of the list of Certicom ECDLP challenge problems have been solved. Escott *et al.* report on their successful attack on ECCp-97, an ECDLP in a group of roughly $2^{97}$ elements [8]. A larger instance, ECC2-109 was solved by Monico *et al.* [6]. Bos *et al.* analyze and solve the ECDLP in a group of roughly $2^{112}$ elements using PS3s [4].

This paper reports on our effort to solve one of the Certicom ECDLP challenge problems using FPGAs. We focus on a Koblitz curve challenge over $\mathbf{F}_{2^{131}}$, called "ECC2K-130." Compared to previous attacks, which are mostly implemented in software on general-purpose workstations, our work obtains a much higher performance-cost ratio by using FPGA platforms.

The rest of the paper is organized as follows. Section II summarizes related work. In Section III we give a short description of the function that we implement on FPGA. Section IV and V explore different algorithms and architectures. Section VI reports on the results, concluding in Section VII.

## II. Related Work

The strongest known attacks against the ECDLP are generic attacks based on Pollard's rho method [20], [5]. Further improvements including parallelization and the use of group automorphisms were made by Wiener and Zuccherato [26], van Oorschot *et al.* [25], and Gallant *et al.* [9]. The parallelized Pollard rho method consists of parallel loops that search for distinguished points. Each loop starts from a random point on $E$ and ends when a distinguished point is hit. The core function is thus the iteration function. Our work implements this function along with its improvements.

FPGAs have been applied to the Pollard rho method in several previous works. Güneysu *et al.* analyze ECDLPs over fields of odd-prime characteristic [11], [10], targeting a machine with 128 low-cost FPGAs. They extrapolate that to break an ECDLP in a group of roughly $2^{131}$ elements using this machine as taking over a thousand years.

Similarly, Meurice de Dormale *et al.* apply FPGAs to the ECDLP [17]. Here, they use characteristic-two finite fields, but restrict their inquiry to polynomial basis. Although conventional wisdom has held that low-weight polynomial basis is a better choice, in our application we can take advantage of the free repeated squarings ($2^n$-th powers) offered in normal basis. In addition, recent progress on normal-basis multiplication by Shokrollahi *et al.* [24] and Bernstein and Lange [2] further improve the prospects for normal basis. Our work is the first

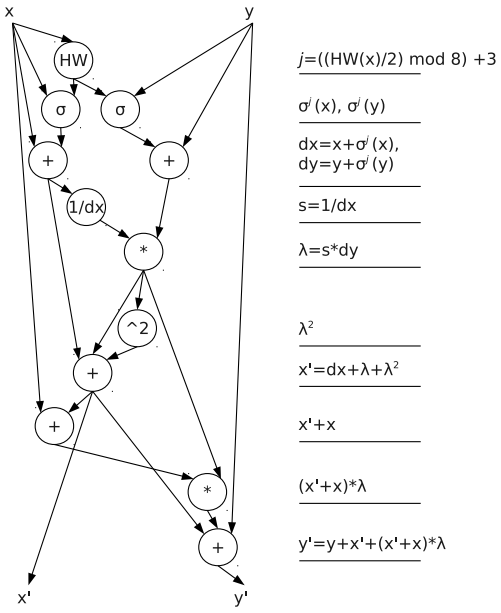


Fig. 1: Dataflow of the iteration function: $P_{i+1} = \sigma^j(P_i) \oplus P_i$. (Note here $\sigma^j(x) = x^{2^j}$)

FPGA implementation of the parallelized Pollard rho method using normal basis multiplication.

This work is part of a global distributed effort to break the largest ECDLP ever solved [1]. While that paper summarizes the overall effort, here we focus on the FPGA implementation.

**The Contribution of This Paper.** As part of this effort, this paper explores FPGA implementation options for the core finite-field arithmetic operations as well as architectures. An in-depth comparison between multipliers (polynomial basis, Type-II normal basis, and Shokrollahi's) is given. Most notably, this is the first FPGA implementation of Shokrollahi's multiplication algorithm. Our work proves the superiority of an FPGA platform over other specialized architectures and its suitability for the tasks that are computationally demanding. Results in this paper are relevant both to the cryptanalytic community as well as those interested in fast cryptographic implementations in normal basis.

## III. Iteration function

We briefly describe the iteration function which we implemented in the FPGAs in this section. The general attack strategy and the design rationale behind of the iteration function can be found in [1].

Our condition for a point $P_i(x, y)$ to be a distinguished point is that $\mathrm{HW}(x) \leq 34$, where $x$ is represented using type-II normal-basis and HW($x$) returns the Hamming weight of $x$. Our iteration function is defined as

$$P_{i+1} = \sigma^j(P_i) \oplus P_i, \tag{1}$$

where $\sigma^j(P_i) = (x^{2^j}, y^{2^j})$ and $j = ((\mathrm{HW}(x_{P_i})/2) \bmod 8) + 3$. To solve ECC2K-130, about $2^{60.9}$ iterations are expected in total [1].

An efficient implementation of the iteration function is thus the key step towards a fast attack. Given $P_i(x, y)$, the iteration function computes $P_{i+1}(x', y')$ using Eq.(1). Fig. 1 shows the data flow of the iteration function.

## IV. Optimizing Finite-Field Operations

The iteration function consists of two multiplications, one inversion and several squarings in $\mathbf{F}_{2^m}$. Thus, fast finite-field arithmetic is essential to optimize the attack.

A vast body of literature exists on finite field arithmetic, and we are free to choose from a variety of representations and algorithms. For example, we can use either polynomial basis or normal basis for element representation, and an iteration can be performed using either the Extended Euclidean Algorithm (EEA) or Fermat's little theorem. This leads to an important question: which configuration (basis, multiplication algoirthm, inversion algorithm) ensures the most efficient implementation of the aforementioned iteration function? We try to answer this question with complexity analysis and design-space exploration.

### A. Multiplication

An element of $\mathbf{F}_{2^{131}}$ can be represented in both polynomial basis **P** and Type-II normal basis **N**, where

$$\mathbf{P} = \{\ 1, w, w^2, \cdots, w^{130}\},$$
$$\mathbf{N} = \{\ \gamma + \gamma^{-1}, \gamma^2 + \gamma^{-2}, \gamma^{2^2} + \gamma^{-2^2}, \cdots, \gamma^{2^{130}} + \gamma^{-2^{130}}\}.$$

Here $w$ is a root of an irreducible polynomial of degree 131, while $\gamma$ is a primitive $263^{rd}$ root of unity. Multiplication in polynomial basis has long been considered more efficient than normal basis. On the other hand, squaring in normal basis is simply a circular shift. Moreover, computing any power $\alpha^{2^n}$ can be performed by circularly-shifting by $n$ positions. We implemented both options for comparison.

Besides conventional multiplication algorithms in polynomial and normal basis, we also implemented a recently reported hybrid algorithm, due to Shokrollahi [24]. When multiplication is needed, two field elements are converted to polynomial basis, a polynomial-basis multiplication is carried out, then the results are converted back to normal basis and reduced. This paper includes the first FPGA implementation of this method.

*1) Polynomial-Basis Multiplier:* Algorithms for multiplication in polynomial basis consist of two steps, polynomial multiplication and modular reduction. They can be carried out separately or interleaved. Given two elements $A(w) = \sum_{i=0}^{m-1} a_i w^i$ and $B(w) = \sum_{i=0}^{m-1} b_i w^i$, a bit-serial modular multiplication algorithm is shown in Alg. 1.

It is well known that one way to reduce area complexity is to use a special-form polynomial $P(w)$. For $\mathbf{F}_{2^{131}}$ there exists a low-weight irreducible pentanomial $P(w) = w^{131} + w^{13} + w^2 + w + 1$. Thus, the complexity of step 3 in Alg. 1 is $(m+4)$ XOR and $(m+4)$ AND operations.

One can also compute $C(w) = A(w)B(w) = \sum_{i=0}^{2m-2} c_i w^i$ first, and then reduce it with $P(w)$. In this case, the Karatsuba method [14] can be used to reduce the complexity of

**Algorithm 1** Bit-serial modular multiplication in $\mathbf{F}_{2^m}$

**Input:** $A(w) = \sum_{i=0}^{m-1} a_i w^i$, $B(w) = \sum_{i=0}^{m-1} b_i w^i$ and $P(w)$.
**Output:** $A(w)B(w) \bmod P(w)$.

```
C(w)(= Σ_{i=0}^{m} c_i w^i) ← 0;
for i = m − 1 to 0 do
    C(w) ← w(C(w) + c_m P(w) + b_i A(w));
end for
```

**Return:** $C(w)/w$.

polynomial multiplication. The reduction phase requires O($m$) AND and XOR operations when low-weight $P(w)$ exists. For example, when $P(w)$ is a pentanomial, reducing $C(w)$ requires around $4m$ AND and $4m$ XOR operations. The overall complexity of a modular multiplication is M($m$)+O($m$), where M($m$) is the complexity of an $m$-bit polynomial multiplication.

*2) Normal-Basis Multiplier:* The normal-basis multiplier by Sunar and Koç [23] employs the fact that an element $(\gamma^{2^i} + \gamma^{2^{-i}})$ for $i \in [1, m]$ can be written as $(\gamma^j + \gamma^{-j})$ for some $j \in [1, m]$. As a result, the following basis **pN** is equivalent to **N**:

$$\mathbf{pN} = \{\gamma + \gamma^{-1}, \gamma^2 + \gamma^{-2}, \gamma^3 + \gamma^{-3}, \cdots, \gamma^{131} + \gamma^{-131}\}.$$

**pN** is also known as permuted normal basis. Let $\beta_i = (\gamma^i + \gamma^{-i})$, then an element $T$ in $\mathbf{F}_{2^m}$ is represented as $T = \sum_{i=1}^{m} t_i \beta_i$. One multiplication of $A$ and $B$ represented with **pN** requires $m^2$ AND and $3m(m-1)/2$ two-input XORs.

This algorithm is then adapted by Kwon [16] to deduce a systolic multiplier. Compared to the Sunar-Koç multiplier, Kwon's architecture (Fig. 3 in [16]) is highly regular and thus can be implemented in a digit-serial manner. On the other hand, it has higher complexity: $2m$ AND and $2m$ XOR gates for a bit-serial multiplier.

*3) Shokrollahi's multiplier:* Shokrollahi discovered an efficient algorithm for basis conversion between permuted normal basis and polynomial basis. Later, Bernstein and Lange proposed further optimizations to this approach including a more straight-forward conversion function. More details on this multiplication method can be found in their recent work [2]. The new polynomial basis (**nP**) is defined in [2] as Type-II polynomial basis.

$$\mathbf{nP} = \{(\gamma + \gamma^{-1}), (\gamma + \gamma^{-1})^2, \cdots, (\gamma + \gamma^{-1})^m\}$$

which leads to a hybrid normal-basis multiplication algorithm. We denote $A_{nP}$ and $A_{pN}$ the representation of $A$ using **nP** and **pN**, respectively. A multiplication then proceeds as follows.

1) converting to polynomial basis: $A_{nP} \leftarrow A_{pN}$, $B_{nP} \leftarrow B_{pN}$,
2) polynomial multiplication: $C_{nP} \leftarrow A_{nP} B_{nP}$,
3) converting back to normal basis ($2m$-bit conversion): $C_{pN} \leftarrow C_{nP}$,
4) reduction (folding).

Let $S_{pN2nP}$ be a transformation function that converts $A_{pN}$ to $A_{nP}$. The essential observation by Shokrollahi is that basis conversion can be recursively broken down to half-length transformations. Let $f_{pN}$ and $f_{nP}$ be corresponding representations of $f$ in **pN** and **nP**, respectively,

$$f_{pN} = [f_1\ f_2\ \cdots\ f_8] \odot [(\gamma + \gamma^{-1})\ (\gamma^2 + \gamma^{-2})\ \cdots\ (\gamma^8 + \gamma^{-8})]^T,$$
$$f_{nP} = [g_1\ g_2\ \cdots\ g_8] \odot [(\gamma + \gamma^{-1})\ (\gamma + \gamma^{-1})^2\ \cdots\ (\gamma + \gamma^{-1})^8]^T,$$

Converting $f_{pN}$ to $f_{nP}$ can be then performed with two 4-bit transformations:

$$\{g_1, g_2, g_3, g_4\}_{\mathbf{nP}} \xleftarrow{S_{pN2nP}} \{f_1 + f_7, f_2 + f_6, f_3 + f_5, f_4\}_{\mathbf{pN}}$$
$$\{g_5, g_2, g_7, g_8\}_{\mathbf{nP}} \xleftarrow{S_{pN2nP}} \{f_5, f_6, f_7, f_8\}_{\mathbf{pN}}$$

The **pN** $\rightarrow$ **nP** and **nP** $\rightarrow$ **pN** conversion in $\mathbf{F}_{2^m}$ takes $(m/2)\log_2(m/4)$ operations each. In total, one field multiplication takes about M($m$) + $m \log_2 m$ + $m \log_2(m/4)$ operations. A more detailed discussion on the complexity can be found in [2].

Based on the analysis above, we can draw the following conclusions:

- A bit-serial multiplier using polynomial basis has a lower area complexity than one using normal basis.
- When low-weight polynomials exist, Shokrollahi's multiplication algorithm is likely to have a higher complexity than convential polynomial basis multiplication since the base conversion step is more complex than the polynomial reduction.

Though it seems that polynomial basis should be used, normal basis offers several advantages in this specific application. First, the Pollard rho iteration function requires the Hamming weight of $x$-coordinate represented in normal basis. In fact, thanks to the Frobenius endomorphism, checking HW of $x$ in normal basis checks 131 points simultaneously. This brings a speedup of $\sqrt{131}$ to the attack [1]. Second, the iteration function includes two $\sigma^j(x) = x^{2^j}$ routines (known as $m$-squaring). In normal basis, $\sigma$ is essentially a circular shift of $j$ bits, and thus can be performed in one cycle. Gains in $m$-squaring compensate the loss in multiplications.

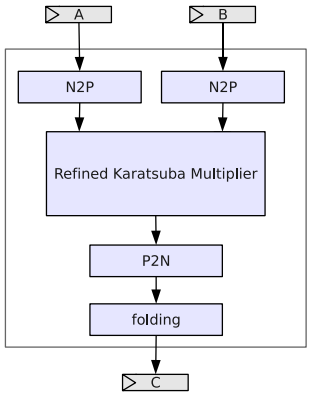


Fig. 2: Shokrollahi multiplier

### B. Inversion

Inversion is the most costly of the four basic field operations. The Extended Euclidean Algorithm (EEA) and Fermat's

**Algorithm 2** Simultaneous inversion (Batch size = 3)

**Input:** $\alpha_1, \alpha_2, \alpha_3$.
**Output:** $\alpha_1^{-1}$, $\alpha_2^{-1}$ and $\alpha_3^{-1}$.

1: $d_1 \leftarrow \alpha_1$
2: $d_2 \leftarrow d_1\alpha_2$
3: $d_3 \leftarrow d_2\alpha_3$
4: $u \leftarrow d_3^{-1}$
5: $t_3 \leftarrow ud_2$, $u \leftarrow u\alpha_3$
6: $t_2 \leftarrow ud_1$, $u \leftarrow u\alpha_2$
7: $t_1 \leftarrow u$

**Return:** $t_1, t_2, t_3$.

Little Theorem (FLT) are widely used to perform inversion. In polynomial basis, the binary variant of EEA is generally faster than FLT, but it requires dedicated hardware. On the other hand, FLT can be performed on standard multiplier, thus has almost no area overhead. In normal basis, the variant of FLT attributed to Itoh and Tsujii [13] is a better choice since squaring is free.

To further reduce the computation costs for inverses, we employ Montgomery's trick that batches multiple inversions by trading inversions for multiplications [19]. Alg.2 shows this method to invert three inputs. Indeed, we can trade one inversion for three extra multiplications. As a result, one iteration function uses 5M + (1/$n$)I where $n$ is the batch size.

Using this trick, a high performance inverter is no longer needed. Indeed, the delay caused by the inverter is 1/$n$ cycles. It is negligible given a large $n$. Therefore, we choose FLT instead of EEA.

## V. Architecture Exploration

The architecture of the engine has a fundamental impact on the overall throughput. Among all the design options the following two are of great importance.

1) Multiplier architecture
2) Memory architecture

As an architecture exploration, we implemented three different architectures using different types of multipliers.

### A. Architecture I: Polynomial Multiplier

As a starting point, we take a programmable elliptic-curve coprocessor as the platform. A digit-serial polynomial multiplier (see [21] for details) is used. A dedicated squarer is included. In each loop, the $x$-coordinate is converted into its normal basis representation, and its Hamming weight is counted. This adds a base-conversion block and a Hamming-weight computation block.

On this platform, squaring or addition takes two clock cycles, while multiplication takes $\lfloor n/d \rfloor + 1$ cycles given a digit-size $d$. The design is synthesized using ISE 11.2 and the target FPGA is Xilinx Spartan-3 XC3S5000 (4FG676). Implementation results show that $d = 22$ gives the best trade-off in terms of area-delay product.

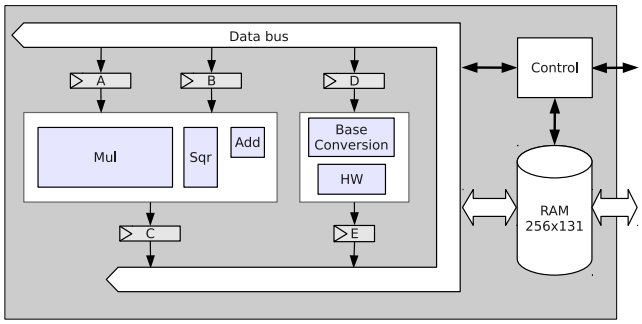


Fig. 3: *Archi-I*: ECC processor using polynomial basis

The design consumes 3,656 slices, including 1,468 slices for the multiplier, 75 slices for the squarer, 1,206 slices for the base conversion, 117 slices for Hamming weight calculation.

One Pollard rho iteration takes 71 cycles; 35 cycles are used for multiplication. The design achieves a maximum clock frequency of 101 MHz, and one iteration takes 704 ns. The $m$-squaring is performed with $m$ successive squarings. Obviously, this architecture is not efficient. The $m$-squaring operations can be largely sped up when normal basis is used.

### B. Architecture II: Type-II Normal Basis Multiplier

*Archi-II* uses a digit-serial normal basis multiplier. When $m$ is small, a full systolic architecture can be used, performing one multiplication per cycle. However, a systolic array for $\mathbf{F}_{2^{131}}$ is too large (more than 20,000 slices on Spartan-3). Thus, a digit-serial architecture is used. Implementation results show that $d = 13$ gives the lowest area-delay product. The multiplier alone uses 2,093 slices.

The basis-conversion component in *Archi-I* is no longer needed in *Archi-II*, saving 1,468 slices. In total, the design uses 2,578 slices. On this platform, one Pollard rho iteration takes 81 cycles, including 55 cycles used for multiplication. Compared to *Archi-I*, the $m$-squaring operation is largely improved. However, the multiplier becomes much slower than that in *Archi-I*. The design achieves a maximum clock frequency of 125 MHz, and one iteration takes 648 ns.

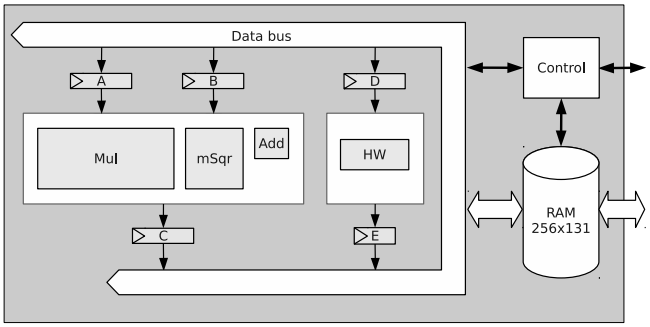


Fig. 4: *Archi-II*: ECC processor using normal basis

### C. Architecture III: Fully Pipelined Iteration Function

*Archi-III* unrolls the Pollard rho iteration such that a throughput of one iteration per cycle is achieved. Remember

that 5 multiplications are required for each iteration, as a result, five normal basis multipliers are used. The design is fully pipelined. Since additions and squarings are embedded in the pipeline, it increases the delay of one iteration but does not affect the throughput.

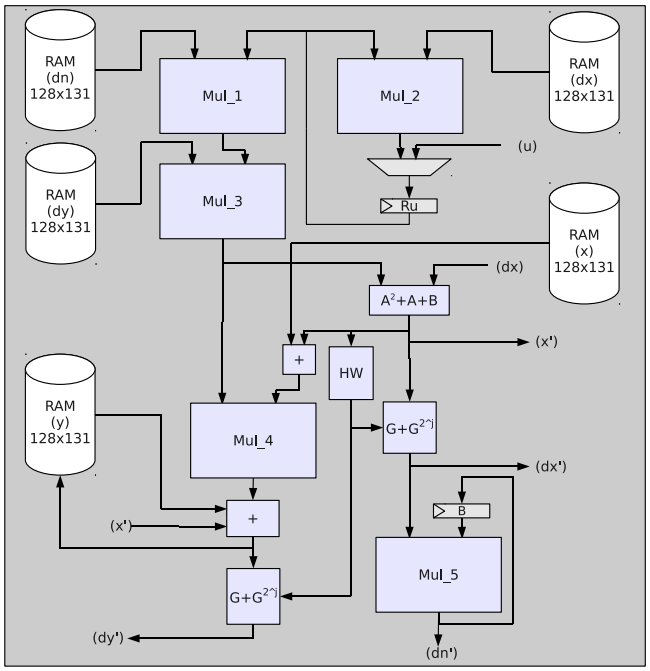


Fig. 5: *Archi-III*: pipelined processor using Shokrollahi multipliers

At the first glance, fully expanding the iteration function seems impossible due to the inversion in each iteration. Indeed, after $dx$ is generated in Fig.1, inverting $dx$ will take too much area to fit in our target FPGA. The solution is to start the pipeline after the real inversion ($u \leftarrow d_n^{-1}$) is performed.

Fig 5 shows the architecture that supports the expanded iteration function. In total, five multipliers are used. Before starting the pipeline, $x$,$y$,$dx$ and $dy$ of $P_i$ are stored in RAM $(x)$, $(y)$, $(dx)$ and $(dy)$, respectively. RAM $(dn)$ keeps the intermediate data $d_i$ of Alg. 2, and $u$ is ready in register $Ru$. After starting the pipeline, the five multipliers perform the following operations.

- Mul_1: $t_i \leftarrow ud_{i-1}$
- Mul_2: $u \leftarrow u\alpha_i$
- Mul_3: $\lambda \leftarrow dy(1/dx)$
- Mul_4: $\lambda(x' + x)$
- Mul_5: $d'_n \leftarrow d'_n dx'$

Mul_1, Mul_2 and Mul_5 are used by batch inversion (Alg. 2), while Mul_3 and Mul_4 are used for point addition (Fig. 1).

The inversion ($u \leftarrow d_n^{-1}$) is performed by another multiplier together with a squarer. In order to maximize engine utilization, we interleave two groups of iteration function. When the engine is executing one group, the inverter is performing inversion for the other group.

The implementation of *Archi-III* consumes 22,195 slices and 20 block RAMs (RAMB16s) on the Xilinx Spartan-3 XC3S5000 FPGA. One fully pipelined Shokrollahi's multiplier uses 4,391 slices. The inverter itself uses 4,761 slices. In total, the design uses 26,731 slices. The post placing-and-routing results show that this design can achieve a maximum clock frequency of 111 MHz.

## VI. Results and Analysis

TABLE I: Cost for the iteration function using various architectures

| | Area | Freq. | Delay | Throughput |
|---|---|---|---|---|
| | #slice /#BRAM | [MHz] | [Cycles] | per FPGA |
| *Archi-I*: Polynomial basis | 3,656 / 4 | 101 | 71 | $12.8 \times 10^6$ |
| *Archi-II*: Type-II ONB | 2,578 / 4 | 125 | 81 | $18.5 \times 10^6$ |
| *Archi-III*: Shokrollahi's | 26,731 / 20 | 111 | 23 * | $111 \times 10^6$ |

* Note that *Archi-III* has 23 stages in the pipeline, thus has a delay of 23 cycles. But the averge throughput is one iteration per cycle.

The ECC2K-130 attack using FPGAs is conducted using an improved version of the COPACOBANA cluster described in [10], [12], also known as RIVYERA [22]. It is populated with 128 Spartan-3 XC3S5000 FPGAs and an optional 32MB memory per FPGA combined in one 19" housing. All FPGAs are connected with two opposite directed, systolic ring networks that directly interface with the PC (which is integrated in the same housing) via two individual PCI Express communication controllers. Although this setup can obviously provide a significant amount of bandwidth due to its local communication paths, the ECC2K-130 attack design actually requires only moderate communication performance.

Table I summarizes the implementation results on a Spartan-3 XC3S5000 FPGA. Based on the available resources (33,280 slices and 104 BRAMs) of each XC3S5000 FPGA, we also estimated that at most 9 clones of *Archi-I* or 12 clones of *Archi-II* can be implemented on a single FPGA. For *Archi-III*, one clone uses 80% of the available resources of one FPGA.

The throughput per engine, $T_e$ is computed as $T_e = \frac{Freq.}{Cycles\ per\ step}$, and the throughput per FPGA $T_c$ is computed as $T_c = T_e * l$. Here, $l$ is the number of engines on a single FPGA. Compared with *Archi-I*, *Archi-II* has smaller area and shorter delay. In other words, Type-II optimal normal basis has significant advantages for this application. On the other hand, *Archi-III* achieves a 8.6 times speedup over *Archi-II*. The improvement mainly comes from the efficient field arithmetic and the special architecture. The use of Shokrollahi's algorithm significantly improved the throughput of a multiplier, while the expansion of the iteration function hides delays caused by addition and squaring in the pipeline.

In Table II we compare our results with similar implementations on different platforms.

To our knowledge, this is the first FPGA implementation using fast normal-basis multiplication to attack ECDLP. As a point of comparison, we look into the work of Meurice de

TABLE II: Performance comparison

| Source | Platform | Challenge | Frq. [MHz] | Throughput [$\times 10^6$] |
|---|---|---|---|---|
| [17] | Xilinx FPGA S3E1200-4 | ECC2-131 | 100 | 10.0 |
| [1] | Cell CPU 6 SPEs, 1 PPE | ECC2K-130 | 3,200 | 27.7 |
| [1] | Graphics Card GTX 295 | ECC2K-130 | 1,242 | 54.0 |
| [1] | Core 2 Extreme Q6850, 4 cores | ECC2K-130 | 3,000 | 22.5 |
| This work (*Archi-III*) | Xilinx FPGA XC3S5000 | ECC2K-130 | 111 | 111 |

Dormale, et al. [17]. They do not specifically target Koblitz curves and they are using different FPGAs, which makes a fair comparison difficult.

On the other hand, there is an interesting comparison between implementations of the same attack (and thus iteration function) on different platforms other than FPGAs. As part of the global distributed effort to attack ECC2K-130 [1], efforts have been made to speed-up the iteration function on CPUs, GPUs and PlayStation 3. These platforms are state-of-the-art processors supporting massive parallelism. Compared to these platforms, our FPGA implementation is at least 2 times faster in terms of throughput.

The whole complexity of this attack is around $2^{60.9}$ iterations. Populated with 128 FPGA each, a single COPACOBANA finishes $2^{58.7} = 128 \cdot (111 \cdot 2^{20} \cdot 3600 \cdot 24 \cdot 365)$ iterations in one year. We estimate that given five COPACOBANA clusters, the ECC2K-130 challenge can be solved in one year.

## VII. Conclusion

A new efficiency record for FPGAs in cracking public-key cryptosystems based on elliptic curves is reported. We conduct a detailed comparison of different architectures for normal-basis multipliers suited this application. The comparison includes the first FPGA implementation using normal basis. Our results show that even low-cost FPGAs outperform CPUs, the PlayStation 3 platform and even GPUs.

## Acknowledgements

This work was supported in part by K.U. Leuven-BOF (OT/06/40), by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy), by FWO project G.0300.07, and by the European Commission through the ICT programme under contract ICT-2007-216676 ECRYPT II.

The authors would also like to thank Daniel J. Bernstein and Tanja Lange for insightful discussions and suggestions.

## References

[1] D. V. Bailey, L. Batina, D. J. Bernstein, P. Birkner, J. W. Bos, H. Chen, C. Cheng, G. van Damme, G. de Meulenaer, L. J. D. Perez, J. Fan, T. Güneysu, F. Gurkaynak, T. Kleinjung, T. Lange, N. Mentens, R. Niederhagen, C. Paar, F. Regazzoni, P. Schwabe, L. Uhsadel, A. Van Herrewege, and B. Yang. Breaking ECC2K-130. Cryptology ePrint Archive, Report 2009/541, 2009. http://eprint.iacr.org/.

[2] D.J. Bernstein and T. Lange. Type-II Optimal Polynomial Bases. to appear in WAIFI 2010 - http://binary.cr.yp.to/opb-20100209.pdf.

[3] I. Blake, G. Seroussi, and N. Smart. *Advances in Elliptic Curve Cryptography (London Mathematical Society Lecture Note Series)*. Cambridge University Press, New York, NY, USA, 2005.

[4] J. W. Bos, M. E. Kaihara, T. Kleinjung, A. K. Lenstra, and P. L. Montgomery. On the security of 1024-bit RSA and 160-bit elliptic curve cryptography: version 2.1. Cryptology ePrint Archive, Report 2009/389, 2009. http://eprint.iacr.org/2009/389.

[5] R. P. Brent and J. M. Pollard. Factorization of the eighth Fermat number. *Mathematics of Computation*, 36:627–630, 1981.

[6] Certicom. Certicom ECC Challenge. http://www.certicom.com/images/pdfs/cert_ecc_challenge.pdf, 1997.

[7] Certicom. ECC Curves List. http://www.certicom.com/index.php/curves-list, 1997.

[8] A. Escott, J. Sager, A. Selkirk, and D. Tsapakidis. Attacking elliptic curve cryptosystems using the parallel Pollard rho method. *CryptoBytes (The technical newsletter of RSA Laboratories)*, 4:15–19, 1998.

[9] R. P. Gallant, R. J. Lambert, and S. A. Vanstone. Improving the parallelized Pollard lambda search on anomalous binary curves. *Mathematics of Computation*, 69(232):1699–1705, 2000.

[10] T. Güneysu, T. Kasper, M. Novotný, C. Paar, and A. Rupp. Cryptanalysis with COPACOBANA. *IEEE Transactions on Computers*, 57(11):1498–1513, November 2008.

[11] T. Güneysu, C. Paar, and J. Pelzl. Attacking elliptic curve cryptosystems with special-purpose hardware. In *Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, page 215. ACM, 2007.

[12] T. Güneysu, G. Pfeiffer, C. Paar, and M. Schimmler. Three Years of Evolution: Cryptanalysis with COPACOBANA. In *Workshop on Special-purpose Hardware for Attacking Cryptographic Systems - SHARCS 2009*, September 9-10 2009.

[13] T. Itoh and S. Tsujii. A Fast Algorithm for Computing Multiplicative Inverses in $GF(2^m)$ Using Normal Bases. *Inf. Comput.*, 78(3):171–177, 1988.

[14] A. Karatsuba and Y. Ofman. Multiplication of many-digital numbers by automatic computers. 145:595–596, 7 1963.

[15] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987.

[16] S. Kwon. A low complexity and a low latency bit parallel systolic multiplier over GF($2^m$) using an optimal normal basis of type II. In *IEEE Symposium on Computer Arithmetic - ARITH-16*, pages 196–202, 2003.

[17] G. Meurice de Dormale, P. Bulens, and J. J. Quisquater. Collision Search for Elliptic Curve Discrete Logarithm over GF($2^m$) with FPGA. In *Workshop on Cryptographic Hardware and Embedded Systems (CHES 2007)*, pages 378–393. Springer, 2007.

[18] V.S. Miller. Use of elliptic curves in cryptography. In *Advances in Cryptology CRYPTO 85 Proceedings*, pages 417–426, 1986.

[19] P. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48:243–264, 1987.

[20] J. M. Pollard. Monte Carlo methods for index computation (mod $p$). *Mathematics of Computation*, 32:918–924, 1978.

[21] K. Sakiyama, L. Batina, B. Preneel, and I. Verbauwhede. Multicore Curve-Based Cryptoprocessor with Reconfigurable Modular Arithmetic Logic Units over GF($2^n$). *IEEE Trans. Computers*, 56(9):1269–1282, 2007.

[22] SciEngines GmbH. RIVYERA S3-5000, 2010. http://www.sciengines.com/joomla/index.php?option=com_content&view=article&id=60&Itemid=74.

[23] B. Sunar and Ç.K. Koç. An Efficient Optimal Normal Basis Type II Multiplier. *IEEE Transactions on Computers*, 50:83–87, 2001.

[24] J. v. z. Gathen, A. Shokrollahi, and J. Shokrollahi. Efficient multiplication using type 2 optimal normal bases. In Claude Carlet and Berk Sunar, editors, *WAIFI*, volume 4547 of *Lecture Notes in Computer Science*, pages 55–68. Springer, 2007.

[25] P. C. van Oorschot and M. J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12(1):1–28, 1999.

[26] M. J. Wiener and R. J. Zuccherato. Faster attacks on elliptic curve cryptosystems. In *Selected Areas in Cryptography*, volume 1556 of *LNCS*, pages 190–200, 1998.